INTL-0662-US
(P12628)

APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: EXECUTING IRREGULAR PARALLEL CONTROL STRUCTURES

INVENTOR: PAUL M. PETERSEN

Express Mail No. EL911617439US

Date: November 16, 2001

EXECUTING IRREGULAR PARALLEL CONTROL STRUCTURES

Background

The present invention relates to parallel computer systems and, more particularly, allocating work to a plurality of execution threads.

In order the achieve high performance execution of difficult and complex programs, for many years, scientists, engineers, and independent software vendors have turned to parallel processing computers and applications. Parallel processing computers typically use multiple processors to execute programs in a parallel fashion which typically produces results faster than if the programs were executed on a single processor.

In order to focus industry research and development, a number of companies and groups have banded together to form industry sponsored consortiums to advance or promote certain standards relating to parallel processing. The Open Multi-Processing ("OpenMP") standard is one such standard that has been developed. OpenMP is a specification for programming shared memory multiprocessor computers (SMP).

One reason that OpenMP has been successful is due to its applicability to array based Fortran applications. In the case of Fortran programs, the identification of computationally intensive loops has been straightforward, and in many important cases, significant improvements in executing Fortran code on multiprocessor platforms has been readily obtained.

However, the use of the OpenMP architecture for applications, which are not Fortran based, has been much slower to gain acceptance. Typically, that is because these applications are not array based and do not easily lend themselves to being parallelized by programs such as compilers which were originally released for the OpenMP standard.

To address this issue, extensions to the OpenMP standard have been proposed and developed. Once such extension is the OpenMP workqueuing model. By utilizing the workqueuing extension model, programmers are able to parallelize a large number of preexisting programs that previously would have required a significant amount of restructuring.

To support this extension to OpenMP, a new concept of "work stealing" was developed. The work stealing model was designed to allow any thread to execute any task on any queue, which was created in a workqueue structure. Work stealing permits all threads started by a run time system to stay busy even when their particular tasks are finished executing.

The concept of work stealing is central to implementing workqueuing in an efficient manner. However, the original implementations of the work stealing concept, while a tremendous advancement in the art, were not optimized. As such, users were not able to fully realize the potential advantages provided by the workqueuing and work stealing concepts.

Therefore, there is still a significant need for a more efficient implementation of the work stealing model.

Brief Description of the Drawings

Figure 1 is a flow chart of the program flow from source code to an initial thread activation list for a plurality of threads in accordance with one embodiment of the present invention.

Figure 2 illustrates an overview of an algorithm for thread workflow in accordance with one embodiment of the present invention.

Figure 3 illustrates nested taskq structures in accordance with one embodiment of the present invention.

Figure 4 illustrates a flow chart for executing a taskq function in accordance with one embodiment of the present invention.

Figure 5 illustrates a flow chart for a work steal process in accordance with one embodiment of the present invention.

Figure 6 is a schematic depiction of a processor-based system in accordance with one embodiment of the present invention.

Detailed Description

In one embodiment of a computer system according to the present invention, a computer system takes as its input a parallel computer program that may be written in a common programming language. The input program may be converted to parallel form by annotating a corresponding sequential computer program with directives according to a parallelism specification such as OpenMP. These annotations designate, parallel regions of execution that may be executed by one or more threads, as well as how various program variables should be treated in parallel regions. The parallelism specification comprises a set of directives such as the directive "taskq" which will be explained in more detail below.

Any sequential regions, between parallel regions, are executed by a single thread. The transition from parallel execution to serial execution at the end of parallel region is similar to the transition on entry to a "taskq" construct. However, when transitioning out of a parallel region, the worker threads become idle, but when entering a "taskq" region, the worker threads become available for work stealing.

Typically, parallel regions may execute on different threads that may run on different physical processors in a parallel computer system, with one thread per processor. However, in some embodiments, multiple threads may execute on a single processor or vice versa.

To aid in understanding embodiments, a description of the taskq directive is as follows:

Logically, a taskq directive causes an empty queue of tasks to be created. The code inside a taskq block is executed single threaded. Any directives encountered while executing a taskq block are associated with that taskq. The unit of work ("task") is logically enqueued on the queue created associated with the taskq construct and is logically dequeued and executed by any thread. A taskq task may be considered a task-generating task as described below.

Taskq directives may be nested, within another taskq block in which case a subordinate queue is created. The queues created logically form a tree structure that mirrors the dynamic nesting relationships of the taskq directives. The whole structure of queues resembles a logical tree of queues, where the root of the tree corresponds to the outermost task queue block, and the internal nodes are taskq blocks encountered dynamically inside a taskq or task block.

Referring now to Figure 1, an input to the computer system 610 is the source code 101 which may be a parallel computer program written in a programming language such as, by way of example only, Fortran 90. However, the source code 101 may be written in other programming languages such a C or C++ as two examples. This program 101 may have been parallelized by annotating a corresponding sequential computer program with appropriate parallelizing directives. Alternatively, in some embodiments, source code 101 may be written in parallel format in the first instance.

The source code 101 may provide an input into a compiler 103 which compiles the source code into object code and may link the object code to an appropriate run time library, not shown. The resultant object code may be split into multiple execution segments such as 107, 109, and 111. These segments 107, 109, and 111 contain, among other instructions and directives, taskq instances that were detected in the source code 101.

The execution segments 107, 109 and 111 may be scheduled by scheduler 105 to be run on an owner thread of which 113, 115 and 117 are representative. As mentioned above, each of these threads may be run on individual processors, run on the same processor, or a combination of both.

Individual threads 113, 115, and 117 may begin to generate tasks, which may be stored in activation lists 119, 121, and 123, respectively, by executing taskq tasks in the execution segments.

Figure 2 illustrates an overview flow chart of a process a particular thread goes through to generate tasks inside a taskq construct according to one embodiment of the invention. An owner thread, such as 113, 115 or 117 may begin to execute a taskq construct beginning at block 201.

Once the owner thread has entered a taskq construct, the thread may determine whether there are more tasks to generate, block 203. If more tasks are available to generate, then the thread may then generate a task, block 205, that is added to a task queue, block 207, such as illustrated in Figure 3 (303, 309).

After a task is added to a task queue, a determination may be made, block 209, as to whether the thread should continue to execute the taskq construct. If execution is to continue, execution flow may return to block 203 in some embodiments. Otherwise, the thread may save its persistent state information and exit the routine. If at block 203 a determination is made that there are no more tasks to be generated in the taskq construct, then the subroutine may be exited at block 211.

A taskq construct is reentrant and the construct may be entered and exited multiple times as required. To provide for reentrance, a thread may remember where it was when it left execution of the construct and may start execution at the same place when execution of the construct is called for again. This may be accomplished by storing persistent state variables as required. Should a new thread subsequently execute the same taskq construct, the new thread may use the persistent variables stored by the prior thread to begin executing the taskq construct at the same place the prior thread stopped.

Figure 3 illustrates how two stacked taskq constructs (301, 316) and (307, 313) may be nested in some embodiments of the invention. In this example, Taskq construct 307, 313 is nested within the taskq construct 301, 316. While two nested taskq constructs are illustrated, more than two taskq constructs may be nested in some embodiments. Elements 305, 311 and 315 represent other instructions that may be present in the code in some embodiments.

In some embodiments, a taskq task has a task queue associated with it. For example, taskq 301 may have associated with it task queue 303 and taskq 307 may have associated with it task queue 309. Tasks that are generated by the execution of the taskq task 310 structure may be placed in taskq 303. In like manner, tasks generated by the execution of taskq structure 307 may be placed in taskq 309.

In one embodiment of the present invention, a particular thread such as 113, 115, or 117 may own task queue 303 in which case task queue 303 may be part of the thread activation list 119, 121, or 123. For example, if thread 113 owned the taskq structure (301,316), then, the task queue 303 may be owned by thread 113.

Each thread started by the computer system may begin and continue to execute tasks from its own activation list until such time as its activation list is empty of active tasks. A thread without an active task may be considered idle. An idle thread may then go into a work stealing mode, which permits an otherwise idle thread to execute any task on any queue.

Work stealing is an important concept in systems that permit the dynamic creation of nesting of parallelism. Given the typical varying amounts of dynamic parallelism available in different parts of the program and, at different levels of nesting, work stealing may allow a computing system to be considerably more computationally efficient.

Figure 4 illustrates an execution flow chart, which may be used by individual threads. A thread begins execution at block 401 and determines at block 403 whether there is a task available in its local activation stack. This may be determined by a thread walking its local activation stack and looking for work to steal from itself. In other words, the thread determines whether there are any task that the thread may perform in its own activation stack.

If there is a task that it may execute, then that task may be performed by the thread, block 405. After the task is executed, the thread may return to block 403 to determine if there are any other tasks that it can perform from its own activation stack. If no other tasks are found, then the thread may be idle.

To indicate that the thread is now idle, the thread and may lock a data-structure in a central repository, and remove itself from a work flow bit mask. A portion of a bit mask, according to some embodiments, is illustrated in Figure 5.

An idle thread may then go into a work steal mode. In some embodiments, the idle thread gets a copy of a bit mask, block 407, and may copy the bit mask into a local storage area. The thread may then determine if the bit mask is empty, block 409. If the bit mask is empty, the thread may release the lock on the repository and wait for an activation signal, block 411 (thread enters a "wait state").

If a bit mask is not empty, that may mean there may be other tasks that may be performed in some other thread's queue. In some embodiments, the thread releases the lock on the data-structure and then begins a search for a task on another thread's activation queue, block 413.

In one embodiment of the present invention, a thread may search for tasks by inspecting a bit in the bit mask associated with a thread to its right. If the thread adjacent to it on the right does not have its mask bit set, then the thread looks to the next most right bit associated with the next most right thread and so on (modulo N, where N is the number of bits associated with particular threads). In other embodiments, a thread may search the bit mask in a different pattern such as looking at its left most neighbor etc. In still other embodiments, a thread my search the bit mask skipping one or more bits according to a search algorithm.

Once a thread has determined that another thread may have a task that can be executed, the thread may obtain a lock on the activation stack of the thread that has a bit indicating there may be tasks that may be performed, block 415. The thread may then begin to search the locked activation list for a task for it to execute, block 417.

It should be noted that the bit mask is a speculative mechanism. That means, if a bit indicates that a particular thread has a task that may be executed, there may or may not, in fact, be a task that is pending for execution in that particular thread's activation stack.

In block 419, in some embodiments, the thread determines if there is a task available in the locked activation list. Should a thread determine that there is not a task available, that is, the bit mask bit was speculative, then the thread may obtain a lock on the bit mask and clear the bit associated with the thread whose activation list the thread just searched and updates its copy of the bit mask, block 421. Then, in some embodiments, the thread may return to block 409 to search for work to steal.

In some embodiments, if at block 419, the thread determined that a task is available, then the thread releases the lock on the other thread's activation list and executes the task, block 415. If the task executed at block 425 was a taskq task which generates a new taskq task, then the new taskq is assigned to the executing thread and the thread may lock the bit mask, block 429, and may set the bit associated with the activation list from which the new taskq task was assigned if the bit was not already set.

Then, in block 431, the thread may signal to other threads that a task may now be available. The thread then may return to searching its own local activation stack, block 403, to examine its own local activation stack for tasks, etc.

If in block 425 the task executed was not a taskq task, or not a taskq task that generated a new taskq task, in some embodiments, the thread may return to block 403, path B, and begin examining its local activation stack. In other embodiments, the thread may return to block 415, path

C, update its local copy of the bit mask, block 433, and once again search the activation list of the thread from which work was just obtained from.

However, many other possibilities exist. For example, the thread may return to block 407, path D, and once again cycle through the bit mask to find other tasks, which it may execute. In some embodiments, threads that are in a wait state, for example threads waiting at block 411, "wake up" when signaled by a thread in block 431 and begin looking for work that they may steal.

In an embodiment of the present invention, if a thread steals a task from another thread's activation list, and that task is a taskq task, any tasks generated therefrom are stored in the owner's activation list. For example, if thread 115 work steals a task from the activation stack 119 of thread 113, and that task was a taskq task, all tasks generated by the execution of the taskq task by thread 115 are stored in thread 113's activation list 119 and the bit 503 in the bit mask 501 associated with thread 113 is set to indicate that thread 113 may have tasks that other threads can steal.

Referring to Figure 5, in some embodiments, a part of a bit mask 501, which includes three, bits 503, 505 and 507. Bit 503 may be associated with a first thread such as thread 113, bit 505 may be associated with a second thread such as thread 115, and bit 507 may be associated with a third thread such as thread 117. In block 407 and 409 of Figure 4, a thread may obtain a copy of bit mask 501 and examine bits 503, 505 and 507 to see if any of the bits are set. A set bit can be either a one or a zero to depending on the particular system implementation chosen. Of course, the assignment of bits in the bit mask 501 is also implementation specific and may differ from that illustrated. For example, bit 507 may be associated with thread 113 and bit 505 may be associated with thread 117.

As described above, if a thread 115 associated with bit 505 wanted to determine if there was other work to steal, it may examine bit 507 to see if it is set. If that bit is set which indicates that there may be work to steal, then the thread 115 may obtain a lock on thread 117's activation stack as is described in association with Figure 4.

As noted above, the particular search algorithm a thread used to determine if there may be work to steal is implementation specific. However, it may be preferred that the algorithm utilized is one that minimizes the creation of hot spots. A hot spot is where tasks are stolen more often from one thread rather than being evenly distributed among all the threads. The use of a search algorithm that results in a hot spot may sub-optimize the execution of the entire program.

Referring to Figure 6, a processor-based system 610 may include a processor 612 coupled to an interface 614. The interface 614, which may be a bridge, may be coupled to a display 616 or a display controller (not shown) and a system memory 618. The interface 614 may also be coupled to one or more storage devices 622, such as a floppy disk drive or a hard disk drive (HDD) as two examples only.

The storage devices 622 may store a variety of software, including operating system software, compiler software, translator software, linker software, run-time library software, source code and other software.

For the purposes of this specification, the term "machine-readable medium" shall be taken to include any mechanism that provides (i.e., stores and/or transmits) information in a form readable by a machine (e.g., a computer). For example, a machine-readable medium includes, but is not limited to, read only memory (ROM); random access memory (RAM); magnetic disk storage media, optical storage media; flash memory devices.

A basic input/output system (BIOS) memory 624 may also be coupled to the bus 620 in one embodiment. Of course, a wide variety of other processor-based system architectures may be utilized. For example, multi-processor based architectures may be advantageously utilized.

The compiler 103, translator 628 and linker 630, may reside totally or partially within the system memory 618. In some embodiments, the compiler 103, translator 628 and linker 630 may reside partially within the system memory 618 and partially in the storage devices 622.

While the preceding description contains many specifics, these should not be construed as limitations on the scope of the invention, but rather as an exemplification of one or a few embodiments thereof.

While the present invention has been described with respect to a limited number of embodiments, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this present invention.